Adapting the INS3D-LU Code to the CM2 and iPSC/860

Rod Fatoohi*

Report RNR-92-024, August 1992

Abstract

This paper presents the results of parallelizing the INS3D-LU code on a 32k processor Thinking Machines CM2 and a 128 node Intel iPSC/860. The main objective of this work is to study the performance of a full Navier-Stokes solver on a massively parallel SIMD machine and a distributed memory MIMD machine and compare it with its performance on an eight processor Cray Y-MP. The code is based on a Lower-Upper Symmetric-Gauss-Seidel implicit scheme for the pseudocompressibility formulation of the three-dimensional incompressible Navier-Stokes equations. The code was rewritten in CM Fortran with shift operations and run on the CM2 using the slicewise model. Also, it was rewritten with distributed data and Intel message passing calls and run on the iPSC/860. The timing results for two grid sizes are presented and analyzed using both 32 bit and 64 bit arithmetic. Also, the impact of communication and load balancing on the performance of the code is outlined. The results show that reasonable performance can be achieved on these parallel machines. However, the Cray Y-MP has outperformed the CM2 and iPSC/860 for this particular algorithm.

SUBMITTED TO THE JOURNAL OF SUPERCOMPUTING.

*Computer Scientist. NAS Applied Research Branch, NASA Ames, MS T045-1, Moffett Field, CA 94035. The author is an employee of Computer Sciences Co. This work was funded through NASA Contract NAS 2-12961.

1 Introduction

The current generation of parallel computers can perform at peak rates of several billion floating point operations per second. Based on peak rates, some of these machines outperform the fastest vector supercomputer by an order of magnitude. (The peak rate for the 64k processor CM2 is 28.67 Gflops compared to 2.67 Gflops for the eight processor Cray Y-MP.) However, performance of the parallel machines varys significantly from one application to another. Some of them achieve a good percentage of their peak for a wide range of applications while others do that only for very specific well written codes. It is well known that the performance of high speed computers is dependent on several factors including architecture, computational algorithm, and software environment.

One of the major problems with parallel machines is to understand how to use them efficiently. Theoretical studies of this question are valuable. However, comparative studies, wherein the same algorithm is implemented on a number of different architectures, provide an equally valid way to this understanding. Studies, carried out for a wide variety of algorithms and architectures, can highlight features of the architectures and algorithms which make them suitable for high performance. Also, comparative studies can exhibit detailed features of an architecture and/or algorithm which are bottlenecks that are overlooked in theoretical studies. The success of this approach depends on choosing "significant" algorithms for implementation and carrying out the implementation over a wide spectrum of architectures. If the algorithm is trivial or embarrassingly parallel it will fit any architecture well. We need to use algorithms which solve the hard problems attacked in the scientific and engineering community.

This paper presents an experiment where the performance of a full Navier-Stokes solver on a 32k processor Thinking Machines CM2 and a 128 node Intel iPSC/860 is studied. The code, INS3D-LU, is based on a Lower-Upper Symmetric-Gauss-Seidel (LU-SGS) implicit scheme for the pseudocompressibility formulation of the three-dimensional incompressible Navier-Stokes equations [9]. The two machines were chosen because of their architectures, high peak rates, and accessibility. The two machines are distributed memory multiprocessors and represent two different architectures and programming models: a SIMD highly parallel machine and a MIMD moderately parallel machine. Both machines are located at the Numerical Aerodynamics Simulation (NAS) Systems division at NASA Ames Research Center. Performance of the code on the two machines is compared to previous results on eight processor Cray Y-MP [4] which is a shared memory MIMD machine with a small number of processors.

The paper presents the implementation, mapping scheme, results, and simple performance models to understand the impact of certain parameters on the performance of both machines. A brief description of the numerical algorithm is also included for completeness; for more details see [9]. The results are presented using both 64 bit arithmetic (Double Precision abbreviated by DP) and 32 bit arithmetic (Single Precision abbreviated by SP). The paper also describes bottlenecks and limiting factors for algorithms of this class on these architectures.

2 The numerical algorithm

Let t be time; ρ and p the density and pressure; u, v, and w the velocity components in Cartesian coordinates (x, y, z); $\widehat{Q}$ the vector of conserved variables; $\widehat{E}$, $\widehat{F}$, and $\widehat{G}$ the convective flux vectors; and $\widehat{E}_v$, $\widehat{F}_v$, and $\widehat{G}_v$ the flux vectors for the viscous terms. Then the three-dimensional Navier-Stokes equations in generalized curvilinear coordinates $(\xi,\ \eta,\ \zeta)$ can be written as

$$\partial_t \widehat{Q} + \partial_\xi(\widehat{E} - \widehat{E}_v) + \partial_\eta(\widehat{F} - \widehat{F}_v) + \partial_\zeta(\widehat{G} - \widehat{G}_v) = 0 \tag{1}$$

where

$$\widehat{Q} = h\begin{bmatrix} p \\ u \\ v \\ w \end{bmatrix}, \quad \widehat{E} = h\begin{bmatrix} \beta U \\ Uu + \xi_x p \\ Uv + \xi_y p \\ Uw + \xi_z p \end{bmatrix}, \quad \widehat{F} = h\begin{bmatrix} \beta V \\ Vu + \eta_x p \\ Vv + \eta_y p \\ Vw + \eta_z p \end{bmatrix}, \quad \widehat{G} = h\begin{bmatrix} \beta W \\ Wu + \zeta_x p \\ Wv + \zeta_y p \\ Ww + \zeta_z p \end{bmatrix} \tag{2}$$

$$\begin{aligned} \widehat{E}_v &= h[\xi_x E_v + \xi_y F_v + \xi_z G_v] \\ \widehat{F}_v &= h[\eta_x E_v + \eta_y F_v + \eta_z G_v] \\ \widehat{G}_v &= h[\zeta_x E_v + \zeta_y F_v + \zeta_z G_v] \end{aligned} \tag{3}$$

and

$$\begin{aligned} E_v &= \begin{bmatrix} 0 \\ 2\mu\partial_x u - \frac{2}{3}\mu(\partial_x u + \partial_y v + \partial_z w) \\ \mu(\partial_y u + \partial_x v) \\ \mu(\partial_z u + \partial_x w) \end{bmatrix}, \\ F_v &= \begin{bmatrix} 0 \\ \mu(\partial_x v + \partial_y u) \\ 2\mu\partial_y v - \frac{2}{3}\mu(\partial_x u + \partial_y v + \partial_z w) \\ \mu(\partial_z v + \partial_y w) \end{bmatrix}, \\ G_v &= \begin{bmatrix} 0 \\ \mu(\partial_x w + \partial_z u) \\ \mu(\partial_y w + \partial_z v) \\ 2\mu\partial_z w - \frac{2}{3}\mu(\partial_x u + \partial_y v + \partial_z w) \end{bmatrix}, \end{aligned} \tag{4}$$

where β is the pseudocompressibility parameter and h is the cell volume. The contravariant velocity components U, V, and W are defined as

$$U = \xi_x u + \xi_y v + \xi_z w, \quad V = \eta_x u + \eta_y v + \eta_z w, \quad W = \zeta_x u + \zeta_y v + \zeta_z w \tag{5}$$

An unfactored implicit scheme can be obtained from a nonlinear implicit scheme by linearizing the flux vectors about the previous time step and dropping terms of the second and higher order.

$$[I + \alpha\Delta t(D_\xi \widehat{A} + D_\eta \widehat{B} + D_\zeta \widehat{C})]\delta\widehat{Q} = -\Delta t\widehat{R} \tag{6}$$

where $\widehat{R}$ is the residual

$$\widehat{R} = D_\xi(\widehat{E} - \widehat{E}_v) + D_\eta(\widehat{F} - \widehat{F}_v) + D_\zeta(\widehat{G} - \widehat{G}_v) \tag{7}$$

and I is the identity matrix. $\delta\widehat{Q}$ is the correction $\widehat{Q}^{n+1} - \widehat{Q}^n$, where n denotes the time level. D_ξ, D_η, and D_ζ are difference operators that approximate ∂_ξ, ∂_η, and ∂_ζ. $\widehat{A}$, $\widehat{B}$, and $\widehat{C}$ are the Jacobian matrices of the convective flux vectors.

$$\widehat{A} = \frac{\partial \widehat{E}}{\partial \widehat{Q}}, \quad \widehat{B} = \frac{\partial \widehat{F}}{\partial \widehat{Q}}, \quad \widehat{C} = \frac{\partial \widehat{G}}{\partial \widehat{Q}} \tag{8}$$

For $\alpha = \frac{1}{2}$, the scheme is second order accurate in time. For other values of α, the time accuracy drops to first order.

The LU-SGS scheme [9] can be written as

$$\mathcal{L}\mathcal{D}^{-1}\mathcal{U}\delta\widehat{Q} = -\Delta t R \tag{9}$$

where

$$\begin{aligned}
\mathcal{L} &= I + \alpha\Delta t(D_\xi^- \widehat{A}^+ + D_\eta^- \widehat{B}^+ + D_\zeta^- \widehat{C}^+ - \widehat{A}^- - \widehat{B}^- - \widehat{C}^-) \\
\mathcal{D} &= I + \alpha\Delta t(\widehat{A}^+ - \widehat{A}^- + \widehat{B}^+ - \widehat{B}^- + \widehat{C}^+ - \widehat{C}^-) \\
\mathcal{U} &= I + \alpha\Delta t(D_\xi^+ \widehat{A}^- + D_\eta^+ \widehat{B}^- + D_\zeta^+ \widehat{C}^- + \widehat{A}^+ + \widehat{B}^+ + \widehat{C}^+)
\end{aligned} \tag{10}$$

In the framework of the LU-SGS algorithm, a variety of schemes can be developed by different choices of numerical dissipation models and Jacobian matrices of the flux vectors. It is desirable that the matrix should be diagonally dominant to assure the convergence. Jacobian matrices leading to diagonal dominance are constructed so that '+' matrices have nonnegative eigenvalues while '−' matrices have nonpositive eigenvalues. For example,

$$\widehat{A}^\pm = \frac{1}{2}[\widehat{A} \pm \rho(\widehat{A})I] \tag{11}$$

and

$$\rho(\widehat{A}) = \kappa \max[|\, \lambda(\widehat{A})\, |] \tag{12}$$

where $\lambda(\widehat{A})$ represent eigenvalues of Jacobian matrix $\widehat{A}$ and κ is a constant that is greater than or equal to 1. The diagonal matrix of eigenvalues is

$$\widehat{\Lambda}(\widehat{A}) = \begin{bmatrix} U & 0 & 0 & 0 \\ 0 & U & 0 & 0 \\ 0 & 0 & U + C_\xi & 0 \\ 0 & 0 & 0 & U - C_\xi \end{bmatrix} \tag{13}$$

where C_ξ is the pseudospeed of sound.

$$C_\xi = \sqrt{U^2 + \beta(\xi_x^2 + \xi_y^2 + \xi_z^2)} \tag{14}$$

It is interesting to note that the need for block inversions can be eliminated if we use approximate Jacobian matrices of Eq. (11). Setting $\alpha = 1$ and $\Delta t = \infty$ gives a Newton-like iteration. Then, Eq. (10) reduces to

$$\begin{aligned} \mathcal{L} &= \rho I - \hat{A}^{+}_{i-1,j,k} - \hat{B}^{+}_{i,j-1,k} - \hat{C}^{+}_{i,j,k-1} \\ \mathcal{D} &= \rho I \\ \mathcal{U} &= \rho I + \hat{A}^{-}_{i+1,j,k} + \hat{B}^{-}_{i,j+1,k} + \hat{C}^{-}_{i,j,k+1} \end{aligned} \tag{15}$$

where

$$\rho = \rho(\hat{A}) + \rho(\hat{B}) + \rho(\hat{C}) \tag{16}$$

The algorithm permits scalar diagonal inversions since

$$Diagonal(\mathcal{L} \ or \ \mathcal{U}) = \begin{bmatrix} \rho & 0 & 0 & 0 \\ 0 & \rho & 0 & 0 \\ 0 & 0 & \rho & 0 \\ 0 & 0 & 0 & \rho \end{bmatrix} \tag{17}$$

A semidiscrete finite volume method is used for spatial discretization. This method is based on the local flux balance of each mesh cell; for more details see [9].

In order to suppress the tendency for odd and even point decoupling, artificial dissipation models are added to central difference schemes. Dissipation models are often called filters since they work like low pass filters which damp out high frequency modes. The dissipative flux d is explicitly added to the convective flux in the following form.

$$-(d_{i+\frac{1}{2},j,k} - d_{i-\frac{1}{2},j,k} + d_{i,j+\frac{1}{2},k} - d_{i,j-\frac{1}{2},k} + d_{i,j,k+\frac{1}{2}} - d_{i,j,k-\frac{1}{2}}) \tag{18}$$

For simplicity, $d_{i+\frac{1}{2},j,k}$ is denoted by $d_{i+\frac{1}{2}}$ in the following.

In this paper, we use a spectral radius dissipation model which is third order accurate in space. Third order terms formed from fourth differences provide damping.

$$d_{i+\frac{1}{2}} = -\epsilon r(\hat{A})_{i+\frac{1}{2}}(\hat{Q}_{i+2} - 3\hat{Q}_{i+1} + 3\hat{Q}_i - \hat{Q}_{i-1}) \tag{19}$$

where $r(\hat{A})$ is the spectral radius of Jacobian matrix $\hat{A}$ and ϵ is a small constant. Eq. (19) can be rewritten as

$$d_{i+\frac{1}{2}} = -\epsilon r(\hat{A})_{i+\frac{1}{2}}(\hat{T}_{i+1} - 2\hat{T}_i + \hat{T}_{i-1}) \tag{20}$$

where

$$\hat{T}_i = \hat{Q}_{i+1} - \hat{Q}_i \tag{21}$$

3 The mapping scheme

The solution procedure of Eq. (9) consists of the computation of $\hat{R}$, the right hand side (RHS), and the generation of the $\mathcal{L}$ and $\mathcal{U}$ matrices, Eq. (10), and their inversion, the left hand side (LHS). The RHS computation is highly parallel with no data dependency in the three directions. The computation of $\mathcal{L}$ and $\mathcal{U}$ at (i, j, k), as given in Eq. (15), requires the values of $\hat{A}$, $\hat{B}$, and $\hat{C}$ at $(i-1,j,k)$, $(i,j-1,k)$, and $(i,j,k-1)$ for $\mathcal{L}$ and $(i+1,j,k)$, $(i,j+1,k)$, and $(i,j,k+1)$ for $\mathcal{U}$. This means the points of each of the $i+j+k = constant$ hyperplanes can be computed concurrently and they depend on the previous plane for $\mathcal{L}$ and on the next plane for $\mathcal{U}$. This feature of the algorithm can be exploited on parallel machines in different ways depending on the architecture and memory organization. On vector and shared memory architectures this algorithm can be implemented by reordering the three-dimensional arrays in the LHS computation into two-dimensional arrays, that is,

$$\hat{Q}(ipoint, iplane) = \hat{Q}(i,j,k)$$

where *iplane* is the serial number of the plane in the sweep direction, and *ipoint* is the address of the point in that plane. This approach was taken by Yoon et al. [9] in vectorizing the INS3D-LU code and by Fatoohi and Yoon [4] in multitasking this code.

The outlined approach may not be efficient on distributed memory machines because of data locality and memory requirements. Barszcz et al. [1] analyzed different solution algorithms for the triangular solves and introduced a new mapping scheme for distributed memory architectures. The new scheme is called the skew hyperplane mapping and used in this work. This scheme is briefly described here; for more details see [1].

The skew hyperplane mapping is based on packing hyperplanes into two dimensional planes so that they occupy the same volume as the original domain. The m^{th} hyperplane can be defined by the set:

$$H_m = \{(i,j,k), \mid i+j+k = m\}, \quad m = 3,4,...,(N_\xi + N_\eta + N_\zeta),$$

where

$$1 \leq i \leq N_\xi, \quad 1 \leq j \leq N_\eta, \quad 1 \leq k \leq N_\zeta$$

Packing can be accomplished by the following one-to-one mapping:

$$\hat{i} = i,$$
$$\hat{j} = j,$$
$$\hat{k} = ((i+j+k-3) \bmod N_\zeta) + 1$$

At most three hyperplanes can be packed into one $\hat{k}$ plane.

The skew hyperplane mapping can be used either in the LHS computation only or in both the LHS and RHS computations. The first approach requires the transformation of a set of arrays required in the LHS computation and the inverse transformation of another set of arrays required for the next time step. The second approach requires the mapping once and operating on the mapped domain in the computation of both the LHS and RHS. The second approach is adopted in this work.

4 The INS3D-LU code

The flow solver code, INS3D-LU, was developed based on the numerical algorithm described in Section 2. The two major portions of the code are preprocessing and time stepping. There could be a third portion for postprocessing, such as generating output files for plotting the results, but this is not included in this work. The preprocessing portion involves several routines for grid generation, initialization, calculation of cell volumes, and calculation of auxiliary cell volumes and cell center coordinates. The time stepping portion has the following routines:

spec Calculate spectral radii (Eqs. (11) to (13)),

vsflux Calculate viscous flux (Eqs. (3) and (4)),

filtsr Calculate numerical dissipation (Eqs. (18), (20) and (21)),

lusgs Calculate convective flux vectors (Eqs. (2)), Jacobian matrices (Eqs. (8)), and invert matrices (Eqs. (9) and (10)),

bcijk Compute boundary conditions for i,j,k-directions

These five routines constitute one iteration of the LU-SGS scheme.

The INS3D-LU code was used in this study to compute the viscous incompressible flow through a straight square duct. The Reynolds number is 790. The grid generation part is quite simple for this problem. The preprocessing portion of the code takes less than one iteration on one processor of the Cray Y-MP. In this work, we concentrate on the time stepping portion of the code.

5 Multitasking on the Cray Y-MP (Revisited)

The implementation of the INS3D-LU code on eight processors of the Cray Y-MP is summarized here for completeness; for more details see [4]. A profile of the code on a single processor of the Y-MP is given for comparison. The Y-MP located at NAS has eight processors, 128 Mwords of main memory, and 6 nsec clock cycle. Peak performance of the machine is 2.67 Gflops.

The code was vectorized by using two-dimensional arrays of the form given in Section 3; see [9]. It was multitasked by parallelizing the outer loops of the RHS and stripmining the inner loops of the LHS; see [4]. Autotasking was used for multitasking. Table 1 contains the results of multitasking on the Y-MP for two grid sizes. The time per iteration is based on the time stepping portion of the code averaged over ten iterations. The processing rate was obtained by using the hardware performance monitor of the machine. The largest grid achieved a speedup of over seven, compared to single processor results, and a performance rate of about 46% of the peak rate of the machine.

Table 2 shows the timings and performance rates of the main routines of the time stepping portion of the code on a single processor of the Y-MP for the $63 \times 63 \times 63$ domain. These numbers were obtained by using Perftrace of the Y-MP. The performance of these routines ranges between 112 and 244 Mflops.

6 Implementation on the CM2

6.1 The CM2 architecture and software

The CM2 at NAS has 32k 1-bit serial processors, 1024 64-bit Weitek floating point units (FPU), 4 GBytes of memory, a clock rate of 7 MHz, and two front end (FE) machines: a Sun 4/490 and a Vax 6320. The Sun front end machine is used for this work. The FE machines store user programs, perform scalar computations, and send instructions for parallel computations to the processors through one or more sequencers. Regular communication between processors is accomplished through the NEWS (North, East, West, South) grid, where each processor communicates with its neighbors on an n-dimensional grid. The transfer rate between FPUs is about 0.5 MBytes/sec. The peak performance of the machine is 14.34 Gflops.

The Connection Machine Fortran (CMF) compiler [8] has two execution models, fieldwise and slicewise. The fieldwise model uses the single bit processors along with the FPUs to do floating point operations. The slicewise model uses only the FPUs as the basic processing units. In the slicewise model, the CM2 at NAS can be considered as a SIMD machine with 1024 processing elements (PEs). Each PE is a vertex of a ten dimensional hypercube, and has an FPU and 4 MBytes of memory. Each FPU includes pipelined floating point adder and multiplier and several vector and scalar registers. The data path between the PE memory and vector registers is 32 bit wide.

6.2 The implementation

The skew hyperplane mapping scheme was used to adapt the INS3D-LU code to the CM2. The code runs almost entirely on the CM2. Only the generation of the grid and some initialization for the test problem (flow through a straight square duct) were performed on the front end machine. These calculations were performed on the unmapped domain for simplicity and adaptability to solve different problems. Then, two arrays were transformed to the mapped domain and moved to the CM2.

The RHS and LHS computations were performed on the CM using the mapped domain. The RHS computation is done on all grid points of each $\hat{k}$ plane simultaneously since there is no data dependency between hyperplanes of a single $\hat{k}$ plane. (Recall that each $\hat{k}$ plane can contain at most three hyperplanes.) The LHS computation has to be done on the hyperplanes, rather than the $\hat{k}$ planes, since there is a data dependency between hyperplanes.

The mapping scheme implemented on the CM2 stores the $\hat{k}$ dimension in-processor (serial) and partitions the $\hat{i}$ and $\hat{j}$ dimensions across processors (parallel). This means that this mapping has a maximum of $N_\xi \times N_\eta$ parallelism. The CMF compiler partitions an array with parallel dimensions into subgrids based on the array size and the machine size. The parallel dimensions are mapped onto the CM processor array, and then the serial dimension is added. Each subgrid is assigned to a PE, and has an equal number of grid points. The number of grid points per PE is equal to $((N_\xi \times N_\eta)/P) \times N_\zeta$ where P is the number of PEs. Since the vector length of the FPU is four, each subgrid is required to be a multiple of four. In addition, the CMF compiler partitions a routine into basic blocks. Large basic blocks usually mean better performance since that leads to efficient use of the FPUs and their registers.

In order to operate on boundary planes (to impose boundary conditions), boolean masks are used. These masks are generated early in the computation and used in the RHS computation. The LHS computation requires the use of a 3-D boolean mask to mask out the unnecessary points in every $\hat{k}$ plane. This mask has $N_\xi + N_\eta + N_\zeta - 2$ planes; one for every hyperplane. However, the use of the masks implies that some processors are not doing useful work every $\hat{k}$ plane. The current implementation of conditional store operations on the CM2 is not as efficient as the unconditional ones (about three times slower). Therefore, it is more efficient to perform the LHS computation on all grid points in every $\hat{k}$ plane and mask out the unneeded points when computing the residuals at the end of every iteration. This means about three times of the amount of useful work is actually performed in the LHS computation.

One of the features of the LU-SGS algorithm is that points of every hyperplane depend only on the adjacent hyperplanes. The skew hyperplane mapping preserves this feature of the algorithm. This means that all communication in the LHS computation is to nearest neighbors. In the RHS computation, communication is also to nearest neighbors.

6.3 The Results

The results of implementing the INS3D-LU code on the CM2 for two grid sizes are listed in Tables 3 and 4 using Double Precision (DP) and Single Precision (SP). respectively. The CMF compiler release 1.1 operating in the slicewise mode is used to generate these results. Notice that the number of PEs is based on the 64-bit processors rather than the 1-bit serial processors. Memory requirements are based on programmer declared variables, both local and global. Temporary arrays created by the compiler, for optimization and others, are not included in the table. The $127 \times 127 \times 127$ grid requires almost 2 GBytes of memory (at least 512 PEs) to run on the CM2 using DP. Subgrid size reflects the amount of work performed by each PE. All arrays are declared with power-of-two since the grid size ($63 \times 63 \times 63$ or $127 \times 127 \times 127$) reflects the number of cells, rather than the number of grid points. The time per iteration is based on the time stepping portion of the code averaged over ten iterations. Performance rates are based on these timings and the number of flops computed on the Y-MP. The best achieved rates for this code using DP and SP are 231 and 366 Mflops, respectively, which represent 1.6% and 2.6% of the peak performance of the CM2.

Performance of the main routines of the code for several cases is given in Tables 5 through 8. In these tables, execution time per call, percentage of the total execution time for that routine, communication time, percentage of the communication time of the execution time for every routine, and performance rate of that routine are given. (Performance of individual routines for the $127 \times 127 \times 127$ domain is not given.) Communication time is estimated using the TMC Prism programming environment, version 1.0. Only the function *cshift* is used for communication in this code. Performance rate is computed by using the number of flops measured on the Y-MP. Results are given for four cases: 1) $63 \times 63 \times 63$ domain on 256 PEs using DP, 2) $63 \times 63 \times 63$ domain on 512 PEs using DP, 3) $63 \times 63 \times 63$ domain on 256 PEs using SP, and 4)

$127 \times 127 \times 127$ domain on 512 PEs using DP.

Tables 5 through 8 show that the routine *lusgs* takes about 50% of the total execution time of every iteration. The main part of this routine is the LHS computation where the amount of the useful work represents only about one third of the actual work performed on the machine, as described in the previous subsection. The performance rate of this routine, as given in Tables 5 through 7, reflects the useful work rather than the actual work since it is computed based on the number of flops on the Y-MP. If the additional work is included in computing the rate for this routine, its performance would be comparable to *spec* and *vsflux*. These two routines suffer the least deficiency on the CM2.

The communication cost is the main cause for low performance for the routine *filtsr*. This routine computes the numerical dissipation which requires a lot of communication. The routine *bcijk* does very little computation and a lot of data movement and communication on all processors even though only a small fraction of this work is useful. This caused a very low performance rate for this routine on the CM2.

Tables 5 through 8 show that communication time represents a reasonable portion of the total execution time (ranging from 43% to 59% for the four cases) even though all communication is nearest neighbor. This shows that the cost of regular communication is quite significant on the CM2 and should not be ignored in developing codes for the machine.

6.4 Performance analysis

The results given in Tables 3 through 8 show the impact of several parameters on the performance of the code on the CM2. Among these parameters are: number of processors used, domain size, and word length (64 or 32 bits). In order to understand the impact of these parameters, a simple performance model for computation and communication costs is developed. The model for communication costs is derived from the models developed by Levit [7] and Chang [3].

Let $D : (d \times d \times d)$ be a cubic domain and $P : (p_1 \times p_2)$ the number of PEs which partitions D into $G : (g_1 \times g_2 \times g_3)$ subgrids where the first two dimensions of the domain are parallel and the third one is serial so that $g_1 = d/p_1$, $g_2 = d/p_2$ and $g_3 = d$. Then the computation time, T_{comp}, can be modeled by the following simple formula:

$$T_{comp} \propto G \times t_{comp}/4 + C_{comp}, \tag{22}$$

where t_{comp} is the time to do a 64 bit floating point operation on a vector of length four and C_{comp} is the overhead time for the CM to receive addresses and data from the frond end for each floating point vector operation. The value of t_{comp} includes time to load and store operands to memory. The value of C_{comp} depends on the size of the block; i.e, the number of flops in a block. Here C_{comp} is basically the average overhead for each vector operation. For large subgrids and big blocks, C_{comp} could be insignificant. Eq. (22) can be rewritten as

$$T_{comp} \propto (\frac{d^3}{4p_1p_2})t_{comp} + C_{comp} \tag{23}$$

The communication time, T_{comm}, has two components: ON-PE communication time, T_{onpe}, and OFF-PE communication time, T_{offpe}. The ON-PE communication results from moving data within a PE while the OFF-PE communication results from moving data between PEs. These times depend on which dimension data is shifted. For this code, the numbers of shift operations along the first two dimensions are about equal. Also, only communication with distance one is modeled here since all communication is nearest neighbor.

The communication model is based on the observation that a shift with distance one along the first dimension of the array G causes moving a boundary plane of size $g_2 \times g_3$ to its neighboring PE while all other planes, of size $((g_1 - 1) \times g_2 \times g_3)$ total, will move internally within the PE. Similarly, when the array is shifted along the second dimension. This observation is summarized in the following formulas:

$$T_{onpe} \propto (g_1 - 1) \times g_2 \times g_3 \times t_{onpe} + g_1 \times (g_2 - 1) \times g_3 \times t_{onpe} + C_{onpe}, \tag{24}$$

$$T_{offpe} \propto g_2 \times g_3 \times t_{offpe} + g_1 \times g_3 \times t_{offpe} + C_{offpe}, \tag{25}$$

where t_{onpe} is the time to move a 64 bit word within a PE (copying data), C_{onpe} is the startup time in moving data within a PE, t_{offpe} is the time to move a 64 bit word to a neighboring PE, and C_{offpe} is the startup time in moving data externally. This model does not take into account other overheads mentioned by Chang [3]. Eqs. (24) and (25) can be rewritten as

$$T_{onpe} \propto (\frac{d}{p_1} - 1)(\frac{d^2}{p_2})t_{onpe} + (\frac{d}{p_2} - 1)(\frac{d^2}{p_1})t_{onpe} + C_{onpe}, \tag{26}$$

$$T_{offpe} \propto (\frac{d^2}{p_2})t_{offpe} + (\frac{d^2}{p_1})t_{offpe} + C_{offpe}. \tag{27}$$

The CM2 is a SIMD machine where only one instruction can be issued every cycle. This means that at any time the machine can do only one of the following operations: computation, internal communication, or external communication. Therefore, the total communication time is the summation of the internal and external communication times and the total time is the summation of the total communication time and the computation time. The times T_{comp}, T_{onpe}, and T_{offpe} are $O(\frac{d^3}{p^2})$, $O(\frac{d^3}{p^2})$, $O(\frac{d^2}{p})$, respectively, for $p_1 = p_2 = p$.

As previously mentioned, this simple model is developed to understand the impact of certain parameters on the performance of the machine. In order to study the impact of each parameter, that parameter is changed while the other ones remain unchanged. Both the model and the measured timing results are used to understand the impact of each parameter. The model is used mainly to give an upper bound on a possible change while the measured results represent an observed change. Eqs. (23), (26) and (27) will be used in this analysis.

6.4.1 Increasing the number of processors

If the same code (using the same domain) runs on a larger machine, then the subgrid sizes will get smaller; i.e., smaller granularity. So if p_2 is doubled, then changes in the model are given by:

1. T_{comp} will be reduced by up to 50% depending on C_{comp} and t_{comp} (which remain unchanged).

2. C_{onpe} and t_{onpe} will remain unchanged but the first term in Eq. (26) will be reduced by 50% and the second term will be reduced by more than 50%. Therefore, T_{onpe} can be reduced by more than 50% if C_{onpe} is negligible.

3. the first term in Eq. (27) will be reduced by 50% while the other terms will remain unchanged. Therefore, T_{offpe} will be reduced by at most 25% depending on C_{offpe} (assuming that the first two terms in Eq. (27) are equal).

So, the impact of increasing the number of processors on the communication cost depends on which communication cost, ON-PE or OFF-PE, is dominant.

The impact of increasing the number of processors on the performance of the code can be measured from data collected in Tables 5 and 6. Table 9 shows the measured percentage changes in communication, computation, and total time when the number of PEs is increased from 256 to 512. Overall, the computation time is reduced by 42%, the communication time is reduced by only 19%, and the total execution time is reduced by 30%. This means that OFF-PE communication has dominated the communication cost for this code. Also, it means that increasing the number of processors has more impact on the computation time than on the communication time.

If the number of processors is quadrupled, then

1. T_{comp} will be reduced by a factor of up to four.

2. T_{onpe} will be reduced by a factor of about four.

3. T_{offpe} will be reduced by a factor of up to two.

The experimental results (see Tables 3 and 4) shows that for the $63 \times 63 \times 63$ domain when the number of PEs is increased from 256 to 1024 the execution time is reduced by a factor of about two, both in SP and DP. For the $127 \times 127 \times 127$ domain in SP, the execution time is reduced by a factor of 2.8 when the number of PEs is increased by a factor of four. This shows that performance improvement due to increasing the number of processors depends on the subgrid size (an improvement of 2.1 for subgrid $4 \times 4 \times 64$ compared to 2.8 for subgrid $8 \times 8 \times 128$ both in SP).

6.4.2 Reducing the word length

The use of SP (32 bits) instead of DP (64 bits) will have the following impact:

1. t_{comp} will be reduced by at most 50% depending on the locality of variables (vector registers or PE memory) while C_{comp} will be less affected (only in moving scalars to the CM). Therefore, data locality will determine the impact on T_{comp}.

2. t_{onpe} will be reduced by about 50% while C_{onpe} will be less affected. Therefore, T_{onpe} will be reduced by at most 50% depending on C_{onpe}.

3. t_{offpe} will be reduced by 50% while C_{offpe} will be less affected. Therefore, T_{offpe} will be reduced by at most 50% depending on C_{offpe}.

The impact of the word size is measured using data collected from Tables 5 and 7. Table 10 shows the measured reduction in communication, computation, and total times when the code for the $63 \times 63 \times 63$ domain was recompiled and run using 256 PEs and SP. Overall, the communication time was reduced by 46%, the computation time by 24%, and the total time by 35%. This shows that the communication overhead constants, C_{offpe} and C_{onpe}, do not have a major impact on the communication time. Also, it shows that reducing the word length has more impact on the communication time than on the computation time. The results for other cases, i.e., larger number of PEs and larger domain, show a decrease in the total time by 35% to 38%; see Tables 3 and 4. This shows that this parameter is less affected by the other parameters (the number of PEs and domain size).

6.4.3 Increasing the domain size

If the number of grid points in all three dimensions is doubled; i.e., D is increased by a factor of eight, then

1. T_{comp} will be increased by a factor of up to eight depending on C_{comp} and t_{comp} (which remain unchanged).
2. T_{onpe} will be increased by a factor of less than eight depending on the subgrids size and C_{onpe} and t_{onpe} (which remain unchanged).
3. T_{offpe} will be increased by a factor of at most four depending on C_{offpe} and t_{offpe} (which remain unchanged).

The impact of increasing the domain size can be measured using Tables 6 and 8. Table 11 shows the increase in communication, computation, and total time when the grid size is doubled in every dimension while using the same number of processors (512 PEs) and DP. Results show that the communication time is increased by a factor of 3.65 while the computation time is increased by a factor of 6.52. The total time increased by a factor of 4.82. This means that increasing the domain size caused larger increase in the computation time than in the communication time.

Performance improvement resulted from increasing the domain size can be defined by the ratio of the number of processors to the increase in the execution time. Based on this definition, performance improvement of 66% (8/4.82) is obtained for the 512 PEs, DP case. Performance improvement for the 1024 PEs, DP case is about 90% (see Table 3). In SP, performance improvement of 40%, 62%, and 85% using 256, 512, and 1024 PEs, respectively, is obtained (see Table 4). These cases show that performance improvement due to increasing the domain size depends on the size of the subgrid.

7 Implementation on the iPSC/860

7.1 The iPSC/860 architecture and software

The iPSC/860 [5] at NAS has 128 nodes interconnected by a seven-dimensional hypercube network and a front end machine, which is called the System Resource Module (SRM). Each node has a 64-bit i860 microprocessor, 8 MBytes of memory, and a Direct Connect Module (DCM). Each DCM, which is responsible for communication between nodes, has eight channels, seven of which are for hypercube interconnection and one for I/O. Communication between nodes is based on circuit switching where a dedicated path is established between two nodes whenever they want to communicate. The transfer rate between nodes is 2.8 MBytes/sec while the latency for short messages (up to 100 bytes) is 67 μsec [2].

The i860 processor includes two pipelined floating point units, adder and multiplier; an eight KByte data cache; a four KByte instruction cache; an integer unit; and several floating point registers. The clock rate for the machine is 40 MHz. The data path between the floating point registers and the data cache is 128 bit wide while the data path between the registers and local memory is 64 bit wide. In single precision, each of the floating point units (adder and multiplier) can produce one result every clock cycle but in double precision, the adder needs one cycle while the multiplier needs two cycles to produce one result. This yields to a peak performance of 80 Mflops in SP and 60 Mflops in DP for every node. The peak performance of the machine therefore is 10.24 Gflops in SP and 7.68 Gflops in DP.

The programming model for the iPSC/860 is message passing where nodes communicate and synchronize with each other by sending and receiving messages. Calls to deal with messages are added to a serial Fortran code whose data have been partitioned across nodes. Data partitioning as well as inserting communication calls are done by the user.

7.2 The implementation

The skew hyperplane mapping scheme is used to adapt the INS3D-LU code to the iPSC/860. The iPSC/860 version of INS3D-LU has a host code, which runs on the SRM or a remote machine, and a node code, which runs on a set of nodes called a cube. The host code communicates with the node code in sending input data and receiving final results. The host code can be easily eliminated, and its I/O function can be accomplished by one of the nodes. All routines of the code run on the cube. This includes the grid generation and initialization parts which were performed on the unmapped domain.

The domain was partitioned into P subdomains and each subdomain is assigned to a node. Since the LHS computation has only 2-D parallelism, either 1-D or 2-D partitioning schemes can be used. The main limitation with 1-D partitioning is that the number of grid points in the partitioned dimension should be greater than or equal to the number of nodes.

The 2-D partitioning scheme used is implemented by dividing the domain into blocks, *pencils*, along the $\hat{\imath}$ and $\hat{\jmath}$ dimensions; the $\hat{k}$ dimension is stored locally. Each block contains at least $((N_\xi \times N_\eta)/P) \times N_\zeta$ grid points. Each block may also contain two buffers (virtual planes) in each partitioned direction to take into account the interface planes of the four neighboring blocks. These buffers cause an increase in the amount of memory required per node.

In some cases, communication can be eliminated if the information about the interface planes is computed locally. There is a trade-off between computation and communication. If computation is less expensive than communication, as the case with the iPSC/860, then it is always beneficial to do local computation and communicate only when it is absolutely necessary.

The RHS and LHS computations for each block are performed on every node of the iPSC/860 using the mapped domain. Each node performs the RHS computation and communicates with its four neighbors by sending and receiving arrays of data. Most of these arrays are two dimensional, representing sides of each block where the first dimension is either $\hat{\imath}$ or $\hat{\jmath}$ and the second dimension is $\hat{k}$. Some of these arrays are one dimensional ($\hat{k}$ dimension) representing corners of each block. Nodes in general perform the same amount of work in the RHS computation. However, in computing the boundary conditions, especially in *bcijk*, only the boundary nodes do this computation while the rest proceed in their work.

The LHS computation proceed diagonally from one corner of the domain to the opposite corner (for the $\mathcal{L}$ term) and backward (for the $\mathcal{U}$ term). For every hyperplane, each node computes the $\mathcal{L}$ term and sends messages to its east and north neighbors containing the updated interface points. These messages are of variable lengths and packed to reduce the message overhead. The same node also receives messages from the other two neighbors for the next hyperplane. The whole process is repeated in computing the $\mathcal{U}$ term, except that messages are sent to the west and south neighbors and received from the east and north neighbors. This procedure has a load balancing problem since some nodes have more work to do than others.

7.3 The Results

The results of implementing the INS3D-LU code on the iPSC/860 are listed in Tables 12 and 13 using DP and SP, respectively. The Fortran compiler pgftn Unix release 1.3a is used to produce these results. Memory requirements are based on programmer declared variables for all nodes. More storage is required when more nodes are used since the total buffer space increases. The required memory to run the code in DP is about twice as much as for SP since almost all variables are real floating point. The smallest cube that the $63 \times 63 \times 63$ grid can run on should have at least 32 nodes using DP and 16 nodes using SP while the $127 \times 127 \times 127$ grid does not fit in the whole machine using DP. The sizes of the blocks, given in Tables 12 and 13, are based on the average sizes of the arrays for that particular cube since some arrays need a buffer for every neighboring block while others do not. Time per iteration is based on the time stepping portion of the code averaged over ten iterations. Performance rates are based on these timings and the number of flops computed on the Y-MP. The best achieved rates for this code represent less than 2% of the peak performance of the iPSC/860.

Performance of the main routines of the code on 32 nodes of the iPSC/860 using the $63 \times 63 \times 63$ domain and DP is given in Table 14. Execution time for every routine on every node is measured, and the longest time (slowest node) as well as the average time over all nodes are given in the table. The average node results also include the percentage of the total time and the performance rate, based on the number of flops on the Y-MP. The total time for the slowest node is based on the longest time to perform ten iterations; rather than the sum of the maximum times for the main routines. Notice that the total time for the slowest node is the same as the average node since the execution times for these routines overlap; i.e., a node can finish one routine and move to another routine even though other nodes are still working on the first routine. This overlapping is not quite significant (about 3%).

Table 14 shows that the difference between the longest and average times is within 8% except for the routine *bcijk* where there is a factor of over two between the two times. This is because *bcijk* handles boundary conditions and most of the time only boundary nodes perform these operations. The difference between the two times did not show the load balancing problem with the routine *lusgs*, which handles the LHS computation. This is because there is a global sum operation at the end of this routine to compute the maximum residual after every iteration which delays nodes that are ahead. This is the only global communication function in the code.

The main routines of the code have different performance rates for different reasons. The routine *spec*, which calculates the spectral radii, uses an intrinsic function (square root) which has not been implemented efficiently on the iPSC/860. The routine *filtsr*, which calculates the numerical dissipation, does some computation but a lot of communication between nodes. The routine *lusgs* which takes more than 57% of the total time has a load balancing problem, as mentioned before, and causes some nodes to wait for others to complete their work. The routine *bcijk* has a low rate because it does very little computation on a few nodes, however, it causes little harm since it does not consume that much time, less than 1%. The routine *vsflux* suffers the least deficiency on the iPSC/860.

The time to perform the LHS computation within the routine *lusgs* was also measured, since this part behaves differently from the rest of the code. For the above case, $63 \times 63 \times 63$ domain on 32 nodes and using DP, the average and longest times for the LHS computation only are 2.182 seconds and 2.615 seconds, respectively, which represent 36.2% and 43.3% of the total execution time per iteration.

7.4 Performance analysis

A simple performance model is developed in order to understand the impact of certain parameters on the performance of the code on the iPSC/860. These parameters are: number of processors used, domain size, and word length (64 or 32 bits). Certain assumptions are made in this model. For the RHS computation, it is assumed that there is not that much overlapping between the computation time, T_{compr}, and the communication time, T_{commr}, and load is fairly balanced. For the LHS computation, it is assumed that the main issue is load balancing and the communication time is not quite significant since it overlaps with the computation time, T_{compl}. The last observation was also noticed by Barszcz et al. [1]. It is also assumed, for simplicity, that in the RHS computation each block has a buffer for each partitioned dimension even though some blocks have no buffers.

Let $D : (d \times d \times d)$ be a cubic domain and $P : (p_1 \times p_2)$ the number of nodes that partitions D into blocks where the first two dimensions are partitioned across processors and the third one is held in-processor. So that each block is $d/p_1 \times d/p_2 \times d$. Then the RHS computation and communication times can be modeled by the following formulas:

$$T_{compr} \propto (\frac{d}{p_1} + 1)(\frac{d}{p_2} + 1)(d)t_{comp} \tag{28}$$

$$T_{commr} \propto (\frac{d}{p_1})(d)t_{comm} + (\frac{d}{p_2})(d)t_{comm} + C_{comm}, \tag{29}$$

where t_{comp} is the time to perform a floating point operation; t_{comm} is the time to send an eight byte word to a nearest neighbor, which is the only communication in the code; and C_{comm} is the startup time for sending a message of over 100 bytes, since all messages in the RHS are of this size. Bokhari [2] has measured the communication times on the iPSC/860 and found that the values of t_{comp} and C_{comp} are about 177 and 3 μseconds, respectively.

For $p_1 = p_2 = p$, these times are:

$$T_{compr} \propto (\frac{d}{p} + 1)^2 (d)t_{comp} \tag{30}$$

$$T_{commr} \propto (\frac{2d^2}{p})t_{comm} + C_{comm} \tag{31}$$

The model for the LHS computation is given in [1] for $p_1 = p_2 = p$. This model is based on the observation that for most hyperplanes there are nodes with the maximum possible number of points (d^2/p^2) and this determines the speed of the code since these nodes are the slowest. However, for the first and last few hyperplanes, the slowest node has less than d^2/p^2 points. Therefore,

$$T_{compl} \propto [(3d - 2)(\frac{d^2}{p^2}) - 2(\frac{d}{p} - 1)(\frac{d^2}{p^2})]t_{comp}, \tag{32}$$

where the first term represents the maximum number of points per node and the second term represents the reduction in that number for some hyperplanes. This equation can be simplified as

$$T_{compl} \propto (\frac{d^3}{p^2})(3 - \frac{2}{p})t_{comp} \tag{33}$$

The times T_{compr}, T_{commr}, and T_{compl} are $O(\frac{d^3}{p^2})$, $O(\frac{d^2}{p})$, $O(\frac{d^3}{p^2})$, respectively.

In order to understand the impact of the three parameters, only one parameter is changed every time. Both the model and the total execution time are used to understand that impact. Communication time is included in the total time since it could not be measured separately. Eqs. (30), (31) and (33) are used for our analysis.

7.4.1 Increasing the number of nodes

If the number of nodes is quadrupled, then

1. T_{compr} will be reduced by a factor of less than four.
2. T_{commr} will be reduced by a factor of less than two.
3. T_{compl} will be reduced by a factor of less than four.

A case study here is for $63 \times 63 \times 63$ domain in SP, see Table 13, when the number of nodes is increased from 16 to 64, the total execution time is decreased by a factor of 3.1. Experimental results for other cases, both in SP and DP, show factors of 2.8 and 2.9; see Tables 12 and 13.

7.4.2 Reducing the word length

If SP (32 bits) is used instead of DP (64 bits), then

1. depending on the number of adds, and multiplies and the structure of the code, T_{compr} will be reduced by at most 50%.
2. t_{comm} will be reduced by 50% while C_{comm} remains unchanged. Therefore, T_{commr} will be reduced by less than 50%.
3. T_{compl} will be reduced by at most 50%, similar to T_{compr}.

The impact of using SP on the performance of the code is reported in Tables 12 and 13. For the $63 \times 63 \times 63$ domain, using SP instead of DP caused a decrease in the total measured time by 22%, 21%, and 25% using 32, 64, and 128 nodes, respectively.

7.4.3 Increasing the domain size

If the number of grid points in every dimension is doubled, which means that D is increased by a factor of eight, then

1. T_{compr} will be increased by a factor of less than eight.
2. T_{commr} will be increased by a factor of less than four.
3. T_{compl} will be increased by a factor of eight.

The impact of increasing the domain size is measured using SP and 128 nodes (Table 13). The execution time per iteration is increased by a factor of 6.8 when the domain size is increased by a factor of eight. This means that the performance of the code has improved by 18% (8/6.8) with the increase of the domain size.

8 Comparison and Concluding Remarks

8.1 Code performance

The performance of the INS3D-LU code on the CM2 and iPSC/860 is compared to its performance on the Y-MP. Only results in DP are considered here. Table 15 summarizes the results on the three machines, as presented in Tables 1, 3, and 12. The performance rates are compared to single processor rates of the Y-MP. These results show that for the $63 \times 63 \times 63$ domain both machines at NAS could not match the performance of one processor of the Y-MP using this code. Both machines achieve only about one tenth of the performance of the eight processor Y-MP. For the $127 \times 127 \times 127$ domain, the 32k processor CM2 outperformed one processor of the Y-MP by 36%. (Results for this domain on the iPSC/860 are not available for lack of memory.) The code achieved less than 2% of the peak rate of both machines while it achieved over 40% of the peak rate of the Y-MP.

The performance models of the two machines show that the computation times and internal communication time are $O(\frac{d^3}{P})$ while external communication times are $O(\frac{d^2}{\sqrt{P}})$ where d is the number of points in every dimension and P is the number of processors. These models give some explanation for the changes in performance as one parameter is changed. Increasing the number of processors by a factor of four causes performance improvement of two for the smaller domain and 2.8 for the larger domain on the CM2 while it is about three for the smaller domain on the iPSC/860. This shows that increasing the number of processors has a greater impact with large granularity than with smaller ones, since blocks (or subgrids) are larger on the iPSC/860 and for the larger domain on the CM2 than for the smaller domain on the CM2. Performance improvement due to reducing the word length is larger on the CM2 (35% to 38%) than on the iPSC/860 (21% to 25%) mainly because of different microprocessor architectures. Increasing the domain size has a greater impact on the CM2 than on the iPSC/860 also because of task granularity.

The performance models can also be used to explain performance changes if more than one parameter is changed. In addition, the models can be used to predict performance improvement if one or more of the basic timing units is improved, like increasing the network bandwidth or computation speed.

Having a better understanding of the two architectures allows the behavior of the code to be explained.

8.2 Architectural features

The building blocks of the two machines, CM2 and iPSC/860, are different in structure but comparable in performance; especially when the number of these blocks is taken into account. Table 16 shows the main features of the two architectures using DP. The two machines are based on two different microprocessors: Intel i860 and Weitek wtl 3164. These microprocessors feature only one path between the CPUs and local memory, compared to three paths on the Y-MP. This path is used for loading and storing data to and from memory. Also, the add, multiply and load pipelines for these microprocessors run at different rates, while all these pipelines can produce a result every cycle on the Y-MP. Each pipeline has a latency so these rates can be achieved only after an initial delay. The i860 has a cache but unless there is a reuse of data, pipelined load directly from memory is more efficient than through the cache; for more details see [6]. The vector registers of the wtl 3164 are of size four so that all vector operations are based on this size, while vector registers on the Y-MP are of size 64. All these factors — width of memory path, pipeline rates, and latency — contribute to the low performance rates on the CM2 and iPSC/860, compared to the Y-MP. Also, there is a loss in performance due to the communication bandwidth between PEs.

The clock rate and communication bandwidth of the two machines seem to indicate that the iPSC/860 is about four to eight times faster than the CM2 per PE. This means that the two machines at NAS are of comparable performance. Also, the INS3D-LU code achieved comparable performance on the two machines. However, there are software and algorithmic issues that can also affect the performance of the two machines. Other algorithms can exploit certain features of the two architectures in different ways, and therefore, can achieve different performance rates on these two machines.

8.3 Limitations

The adaptation of the INS3D-LU code to the CM2 and iPSC/860 exploited certain features of the algorithm and the machines. The LU-SGS algorithm requires only nearest neighbor communication which is a desirable feature on parallel machines. However, the study showed that even nearest neighbor communication cannot be ignored on these machines, especially on the CM2 where the communication time ranges between 43% and 59% of the execution time. The LHS computation requires the use of hyperplanes where there is performance degradation by a factor of up to three due to either load imbalance, as on the iPSC/860, or computation overhead, as on the CM2. The other limitation is handling boundary conditions where only the boundary processors are active. The impact of boundary conditions is more significant on the CM2 than on the iPSC/860 because of the number of processors and the SIMD nature of the CM2. The iPSC/860 also suffers from inefficient implementation of some intrinsic functions.

There are other issues concerning partitioning and scheduling the work among the processors. Barszcz et al. [1] suggested a multipartitioning scheme where multiple subpartitions are assigned to each processor in order to reduce the load imbalance. They report performance improvement with this scheme in some cases. Other approaches were also considered by Barszcz et al. [1] such as precomputing the Jacobian matrices or using a different partitioning scheme. However, these schemes have other problems like increased memory requirement or limited scalability.

This study concentrated only on power-of-two grid sizes. Other studies (see for example Barszcz et al. [1]) showed that non-power-of-two grids are less efficient on the CM2 because the CMF compiler has to pad the arrays and communication is more expensive with these grids.

The software environment can have a great impact on the performance of the machine, specifically the Fortran compiler and communication functions. The Fortran compiler on these parallel machines is not efficient enough to use some features of the architecture, such as pipelined load on the i860. This factor degrades the performance of these machines even further. Also, communication functions have not been implemented quite efficiently on these machines, such as *cshift* on the CM2.

In summary, this study showed that reasonable performance can be achieved on moderately to highly parallel machines using a full 3-D Navier-Stokes solver. However, the Y-MP has outperformed the CM2 and iPSC/860 for this particular algorithm. This is partly because only a small fraction of the peak performance of the processing elements of these machines can be achieved due to certain architectural features like memory bandwidth.

9 Acknowledgment

I wish to thank Seokkwan Yoon for providing the code and valuable discussions about the algorithm. Also, I am grateful to Eric Barszcz for a critical reading of the manuscript.

References

[1] E. Barszcz, R. Fatoohi, V. Venkatakrishnan, and S. Weeratunga, *Solution of Regular, Sparse Triangular Linear Systems on Vector and Distributed-Memory Multiprocessors* (under preparation).

[2] S. Bokhari, *Communication Overhead on the Intel iPSC/860 Hypercube*, ICASE Interim Report 10, May 1990.

[3] Y. Chang, *Comparison of Finite Difference and the Pseudo-Spectral Approximations for Hyperbolic Equations and Implementation Analysis on Parallel Computer CM-2*, CAM Report 92-02, Dept. of Mathematics, Univ. of California, Los Angeles, January 1992.

[4] R. Fatoohi and S. Yoon, *Multitasking the INS3D-LU code on the Cray Y-MP*, AIAA 10th Computational Fluid Dynamics Conference, Honolulu, June 24-27, 1991, pp. 619 - 626.

[5] Intel Corp., *iPSC/2 and iPSC/860 Users's Guide*, June 1990, Order Number 311532-006.

[6] K. Lee, *On the Floating Point Performance of the i860 Microprocessor*, Report RNR-90-019, NASA Ames Research Center, October 1990.

[7] C. Levit, *Grid Communication on the Connection Machine: Analysis, Performance, and Improvements*, Int. J. High Speed Computing, Vol. 1, No. 2, 1989, pp. 367 - 381.

[8] Thinking Machines Co., *CM Fortran Optimization Notes: Slicewise Model*, Version 1.0, March 1991.

Table 1: INS3D-LU on the Cray Y-MP.

Domain size	Proc	Memory (MBytes)	Time per iteration (sec)	Performance (Mflops)
$63 \times 63 \times 63$	1	121	1.580	167
	2	122	0.833	318
	4	122	0.434	609
	8	122	0.242	1093
$127 \times 127 \times 127$	1	925	13.002	170
	2	930	6.730	328
	4	930	3.395	650
	8	930	1.811	1218

Table 2: Performance of main routines of INS3D-LU on one processor of the Y-MP using $63 \times 63 \times 63$ domain.

Routine	Time per call (sec)	Perc. (%)	Performance (Mflops)
spec	0.101	6.4	244.1
vsflux	0.478	30.4	194.5
filtsr	0.148	9.4	154.5
lusgs	0.844	53.7	146.8
bcijk	0.001	0.1	111.8
Total	1.572	100.0	168.0

[9] S. Yoon, D. Kwak, and L. Chang, *LU-SGS Implicit Algorithm for Three-Dimensional Incompressible Navier-Stokes Equations with Source Term*, AIAA Paper 89-1964-CP, 1989.

Table 3: Performance of INS3D-LU on the CM2 using DP.

Domain size	PEs	Memory (MBytes)	Subgrid size	Time per iteration (sec)	Performance (Mflops)
$63 \times 63 \times 63$	256	228	$4 \times 4 \times 64$	4.519	59
	512	228	$4 \times 2 \times 64$	3.162	84
	1024	228	$2 \times 2 \times 64$	2.264	117
$127 \times 127 \times 127$	512	1824	$8 \times 4 \times 128$	15.104	146
	1024	1824	$4 \times 4 \times 128$	9.540	231

Table 4: Performance of INS3D-LU on the CM2 using SP.

Domain size	PEs	Memory (MBytes)	Subgrid size	Time per iteration (sec)	Performance (Mflops)
$63 \times 63 \times 63$	256	120	$4 \times 4 \times 64$	2.935	90
	512	120	$4 \times 2 \times 64$	1.989	133
	1024	120	$2 \times 2 \times 64$	1.394	190
$127 \times 127 \times 127$	256	960	$8 \times 8 \times 128$	16.828	131
	512	960	$8 \times 4 \times 128$	9.817	225
	1024	960	$4 \times 4 \times 128$	6.027	366

Table 5: Profiling of INS3D-LU on 256 PEs of the CM2 using $63 \times 63 \times 63$ domain and DP.

Routine	Execution time		Communication time		Performance (Mflops)
	Time (sec)	Perc. of total time (%)	Time (sec)	Perc. of exec. time (%)	
spec	0.276	6.1	0.149	54.0	89.3
vsflux	0.833	18.5	0.463	55.6	111.6
filtsr	0.901	20.0	0.640	71.0	25.4
lusgs	2.229	49.4	0.889	39.9	55.6
bcijk	0.273	6.1	0.165	60.0	0.4
Total	4.512	100.0	2.306	51.1	58.6

Table 6: Profiling of INS3D-LU on 512 PEs of the CM2 using 63 × 63 × 63 domain and DP.

Routine	Execution time		Communication time		Performance
	Time (sec)	Perc. of total time (%)	Time (sec)	Perc. of exec. time (%)	(Mflops)
spec	0.201	6.4	0.123	61.2	122.7
vsflux	0.624	19.8	0.381	61.1	149.0
filtsr	0.697	22.1	0.547	78.5	32.8
lusgs	1.455	46.1	0.708	48.7	85.2
bcijk	0.180	5.7	0.114	63.3	0.6
Total	3.157	100.0	1.873	59.3	83.8

Table 7: Profiling of INS3D-LU on 256 PEs of the CM2 using 63 × 63 × 63 domain and SP.

Routine	Execution time		Communication time		Performance
	Time (sec)	Perc. of total time (%)	Time (sec)	Perc. of exec. time (%)	(Mflops)
spec	0.173	5.9	0.083	48.0	142.5
vsflux	0.537	18.3	0.244	45.4	173.1
filtsr	0.529	18.0	0.362	68.4	43.2
lusgs	1.522	51.9	0.483	31.7	81.4
bcijk	0.170	5.8	0.082	48.2	0.7
Total	2.931	100.0	1.254	42.8	90.2

Table 8: Profiling of INS3D-LU on 512 PEs of the CM2 using 127 × 127 × 127 domain and DP.

Routine	Execution time		Communication time	
	Time (sec)	Perc. of total time (%)	Time (sec)	Perc. of exec. time (%)
spec	1.009	6.6	0.415	41.1
vsflux	2.684	17.6	1.480	55.1
filtsr	2.798	18.4	1.865	66.7
lusgs	7.865	51.7	2.587	32.9
bcijk	0.862	5.7	0.493	57.2
Total	15.218	100.0	6.840	44.9

Table 9: Impact of doubling the number of PEs (from 256 to 512) on measured times using 63 × 63 × 63 domain and DP.

Routine	Reduction in communication time (%)	Reduction in computation time (%)	Reduction in total time (%)
spec	17	39	27
vsflux	18	34	25
filtsr	15	42	23
lusgs	20	44	35
bcijk	31	39	34
Total	19	42	30

Table 10: Impact of reducing the word length (from 64 to 32 bits) on measured times using 63 × 63 × 63 domain and 256 PEs.

Routine	Reduction in communication time (%)	Reduction in computation time (%)	Reduction in total time (%)
spec	44	29	37
vsflux	47	21	36
filtsr	43	36	41
lusgs	46	22	32
bcijk	50	19	38
Total	46	24	35

Table 11: Impact of increasing the domain size (from 63 × 63 × 63 to 127 × 127 × 127) on measured times using 512 PEs and DP.

Routine	Increase in communication time	Increase in computation time	Increase in total time
spec	3.37	6.35	5.02
vsflux	3.89	5.28	4.30
filtsr	3.41	6.22	4.01
lusgs	3.65	7.07	5.41
bcijk	4.32	5.59	4.79
Total	3.65	6.52	4.82

Table 12: Performance of INS3D-LU on the iPSC/860 in DP.

Domain size	Nodes	Memory (MBytes)	Block size	Time per iteration (sec)	Performance (Mflops)
$63 \times 63 \times 63$	32	169	$17 \times 9 \times 64$	6.075	44
	64	179	$9 \times 9 \times 64$	3.530	74
	128	200	$9 \times 5 \times 64$	2.166	122

Table 13: Performance of INS3D-LU on the iPSC/860 in SP.

Domain size	Nodes	Memory (MBytes)	Block size	Time per iteration (sec)	Performance (Mflops)
$63 \times 63 \times 63$	16	80	$17 \times 17 \times 64$	8.573	31
	32	85	$17 \times 9 \times 64$	4.727	56
	64	90	$9 \times 9 \times 64$	2.775	95
	128	100	$9 \times 5 \times 64$	1.632	162
$127 \times 127 \times 127$	128	678	$17 \times 9 \times 128$	11.055	200

Table 14: Profiling of INS3D-LU on 32 nodes of the iPSC/860 using $63 \times 63 \times 63$ domain and DP.

Routine	Slowest node	Average node		
	Time (sec)	Time (sec)	Perc. of exec. time (%)	Performance (Mflops)
spec	0.773	0.724	12.0	34.1
vsflux	1.145	1.131	18.7	82.2
filtsr	0.764	0.706	11.7	32.4
lusgs	3.496	3.454	57.3	35.9
bcijk	0.037	0.016	0.3	7.3
Total	6.033	6.033	100.0	43.8

Table 15: Performance comparison of INS3D-LU in DP.

Domain size	Machine	Processors	Ratio to Y-MP/1
$63 \times 63 \times 63$	Y-MP	1	1.00
		2	1.90
		4	3.64
		8	6.53
	CM2	8K	0.35
		16K	0.50
		32K	0.70
	iPSC/860	32	0.26
		64	0.45
		128	0.73
$127 \times 127 \times 127$	Y-MP	1	1.00
		2	1.93
		4	3.83
		8	7.18
	CM2	16K	0.86
		32K	1.36

Table 16: Main features of the CM2 and iPSC/860 in DP.

Feature	CM2	iPSC/860
I. Building Blocks		
Processing Elements (PEs)	wtl 3164	i860
Clock Rate (MHz)	7	40
Memory (MBytes)	4	8
Peak performance (Mflops)	14	60
Width of memory to CPU path	32 bit	64 bit
Add rate (Cycles/result)	1	1
Multiply rate (Cycles/result)	1	2
Read rate (Cycles/result)	2	2
Write rate (Cycles/result)	4	3
II. Machines at NAS		
Number of PEs	1024	128
Memory (GBytes)	4	1
Peak performance (Gflops)	14.34	7.68
Communication between PEs:		
Bandwidth (MBytes/sec)	2.8	0.5
Connectivity	Hypercube	Hypercube